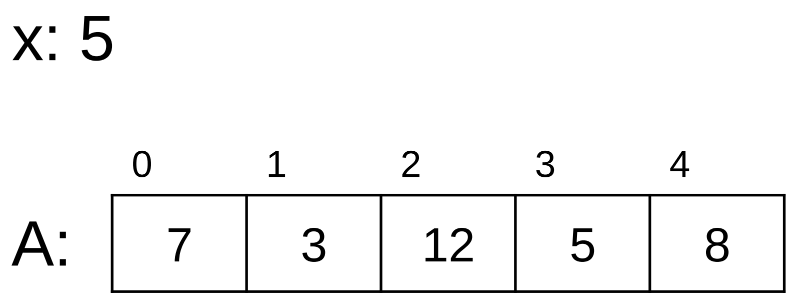
Searching Arrays

Linear Search

Searching for an Element in an Array

- Find where x occurs in A
  - return some index where x appears
  - for x=5, return 3



- **Linear search** algorithm:
  - *look for it in each place until we find it*

- First attempt:

```
int search(int x, int[] A, int n)
{
  for (int i = 0; i < n; i++)
  {
    if (A[i] == x)  return i;
  }
}
```

Searching for an Element in an Array

- Remember **safety**!
  - A[i]: i should be *provably* in bounds
  - n is the length of A

x: 5

	0	1	2	3	4
A:	7	3	12	5	8

- Contracts!

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    if (A[i] == x)  return i;
  }
}
```

Searching for an Element in an Array

- What if x does not occur in A?
  - return something that cannot possibly be an index
  - -1

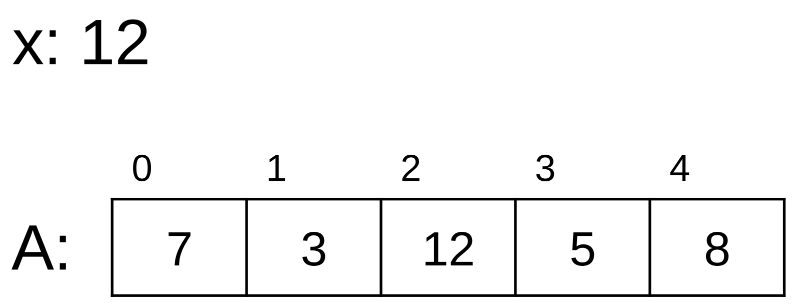
x: 4

	0	1	2	3	4
A:	7	3	12	5	8

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    if (A[i] == x)  return i;
  }
  return -1;
}
```

Searching for an Element in an Array

- How will a caller use search?
  - check if element was in A
    - if returned value is not -1
  - if so, do something with that position
    - e.g., update the value



Caller

```
    …

int k = search(12, A, 5);
 if (k != -1) {
   A[k] = 13; // change 12 to 13
 }
    …
```

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    if (A[i] == x)  return i;
  }
  return -1;
}
```

Searching for an Element in an Array

- How does the caller *know* how search behaves?
  - that -1 is a valid returned value
  - that A[k] contains 12
- Add postconditions!

x: 12

	0	1	2	3	4
A:	7	3	12	5	8

Caller

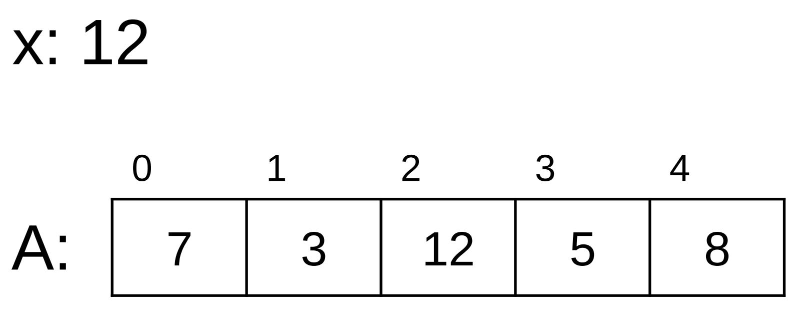
```
    …

int k = search(12, A, 5);
 if (k != -1) {
   A[k] = 13; // change 12 to 13
 }
    …
```

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures \result  == -1
                  || A[\result] == x;
@*/
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    if (A[i] == x)  return i;
  }
  return -1;
}
```

Multiline contract

Searching for an Element in an Array

- Can we be sure that A[\result] is **safe**?
  - Extend postcondition



```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures \result  == -1
              || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    if (A[i] == x)  return i;
  }
  return -1;
}
```

- A[\result] == x won't be called if \result is out of bounds
  - && short-circuits evaluation

Searching for an Element in an Array

- Is search **correct**?
  - *Postconditions are met when preconditions hold*
  - We'll have to prove that

later

x: 12

A:

0	1	2	3	4
7	3	12	5	8

- Does it do what we **expect**?
  - *find x in A*
  - Looks plausible

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures \result  == -1
              || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    if (A[i] == x)  return i;
  }
  return -1;
}
```

Contract Exploits

x: 12

	0	1	2	3	4
A:	7	3	12	5	8

- Is this version of search **correct**?

  Postconditions are met when preconditions hold

  - Definitely!

- Does it do what we **expect**?
  - *find x in A*
  - No!!!!
    - always returns -1

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures \result  == -1
             || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  return -1;          // Always returns -1
}
```

- This is a **contract exploit**
  - Postconditions are met when preconditions hold
    - the function **is** correct
  - but it does not what we expect

Fixing this Contract Exploit

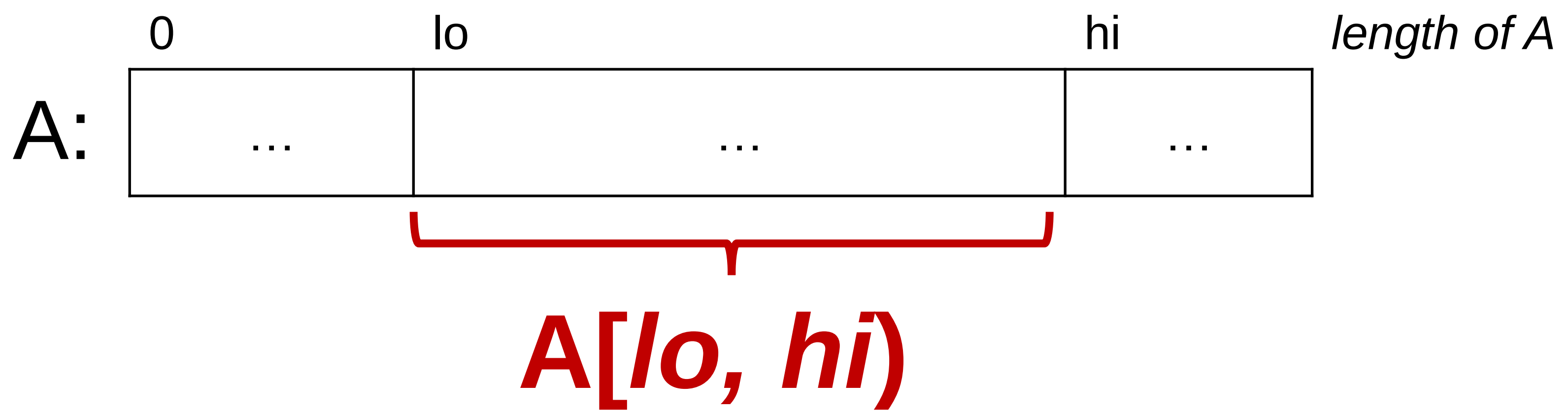
- We want search to return -1 *only if* x does not occur in A
  - Strengthen the postcondition to say just that
  - !is_in(x, A, 0, n)

x: 12

	0	1	2	3	4
A:	7	3	12	5	8

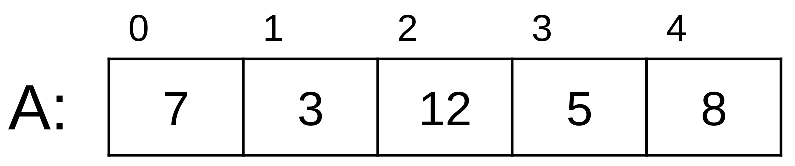
```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                 || (0 <= \result && \result < n && A[\result] == x);
@*/
{
   …
}
```

Array Segments, in Math



Segment of array A between index *lo* included and index *hi* excluded

- Examples:
  - A[1, 4) contains 3, 12, 5
  - A[2, 3) contains 12
  - A[0,5) is the entire array A
  - A[3, 3) does not contain any element: it is an **empty segment**
  - A[4, 2) does not make sense



- we want

$$0 \le lo \le hi \le \textit{length of A}$$

Fixing this Contract Exploit

- Let's define $x \in A[lo, hi)$, in math

$$x \in A[lo, hi) = \begin{cases} \textit{false} & \text{if } lo = hi \\ \text{true} & \text{if } lo \neq hi \text{ and } A[lo] = x \\ x \in A[lo+1, hi) & \text{if } lo \neq hi \text{ and } A[lo] \neq x \end{cases}$$

- Let's implement it as is_in(x, A, lo, hi)
  - This is *specification function*
    - transcription of math
      - obviously correct
      - used interchangeably in proofs
    - meant to be used in contracts
    - often recursive
    - often no postconditions

```
bool is_in(int x, int[] A, int lo, int hi)
//@requires 0 <= lo <= hi <= \length(A);
{
  if (lo == hi)  return false;
  return A[lo] == x || is_in(x, A, lo+1, hi);
}
```

- then, is_in(x, A, 0, n) implements $x \in A[0, n)$
  - is x in the array segment A[0, n)? i.e., is x in A?

Fixing this Contract Exploit

- Fixed code for search

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    if (A[i] == x)  return i;
  }
  return -1;
}
```

- Is it correct?
  - *Postconditions are met when preconditions hold*

Correctness

Correctness

- search has *two* return statements
  - **both** must satisfy the postcondition
- postcondition is a disjunction (||)
  - satisfying one branch is enough

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    if (A[i] == x) return i;
  }
  return -1;
}
```

Correctness (1)

return i on line 10

- **To show**: if $n = \text{\length}(A)$, then
  either $i = -1$ && $x \notin A[0, n)$
  or $0 \le i < n$ && $A[i] = x$

Looks promising

A. $0 \le i$ by line 8

B. $i < n$ by line 7

C. $A[i] = x$ by line 10

✓

```
1.  int search(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  /*@ensures (\result  == -1 && !is_in(x, A, 0, n))
4.                  || (0 <= \result && \result < n && A[\result] == x);
5.  @*/
6.  {
7.    for (int i = 0; i < n; i++)
8.    //@loop_invariant 0 <= i;
9.    {
10.     if (A[i] == x) return i;
11.   }
12.   return -1;
13. }
```

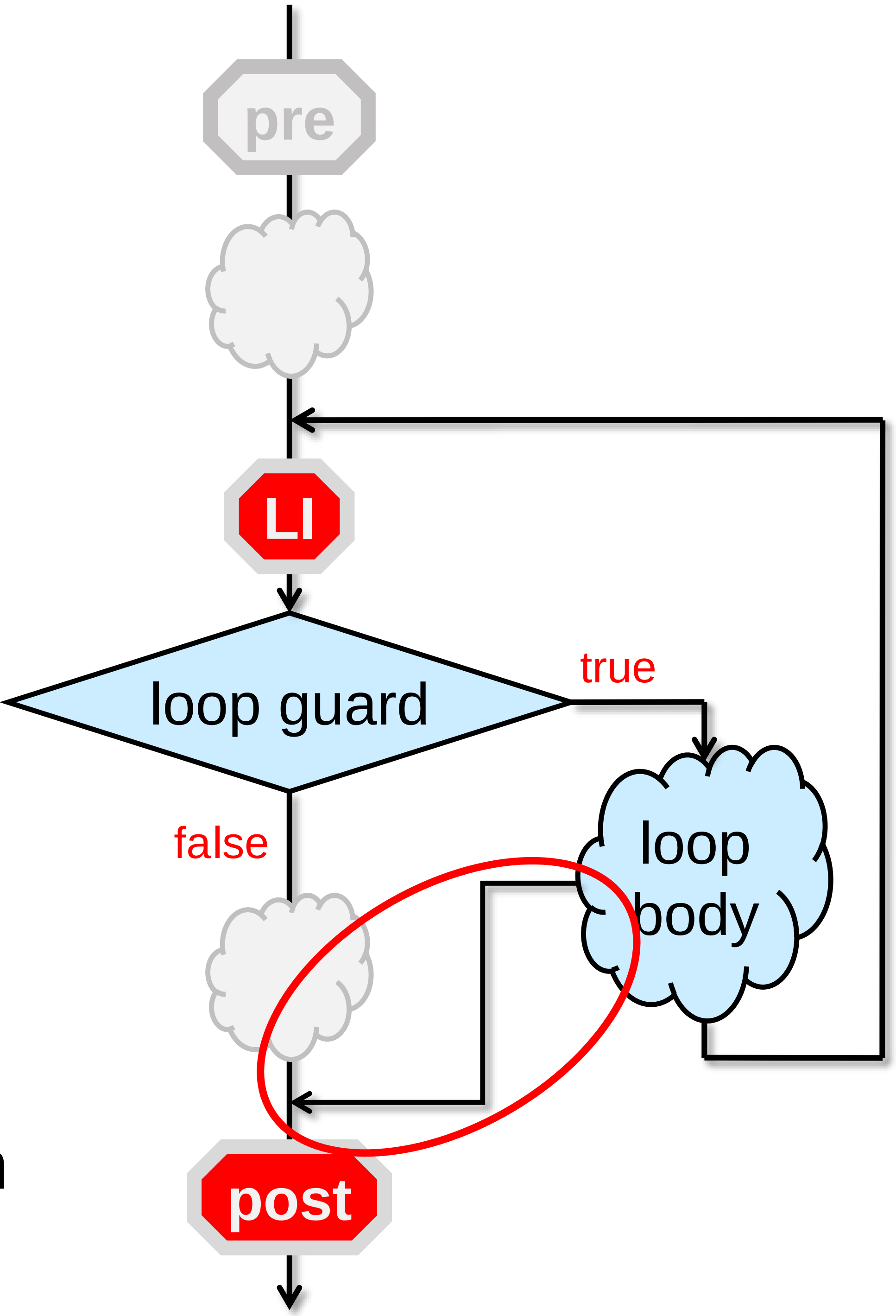
Correctness (1)

return i on line 10

- **To show**: if n = \length(A), then
  either i = -1 && x ∉ A[0, n)
  or 0 ≤ i < n && A[i] = x

A. 0 ≤ i	by line 8
B. i < n	by line 7
C. A[i] = x	by line 10

- We did not use **EXIT**
  - when we return inside the loop,
    we the loop invariant is not checked again



Correctness (2)

return -1 on line 13

- **To show**: if n = \length(A), then

  either $i = -1$ && $x \notin A[0, n)$ — Must be this one

  or $0 \le i < n$ && $A[i] = x$ — *(makes no sense)*

- We must prove

  $x \notin A[0, n)$

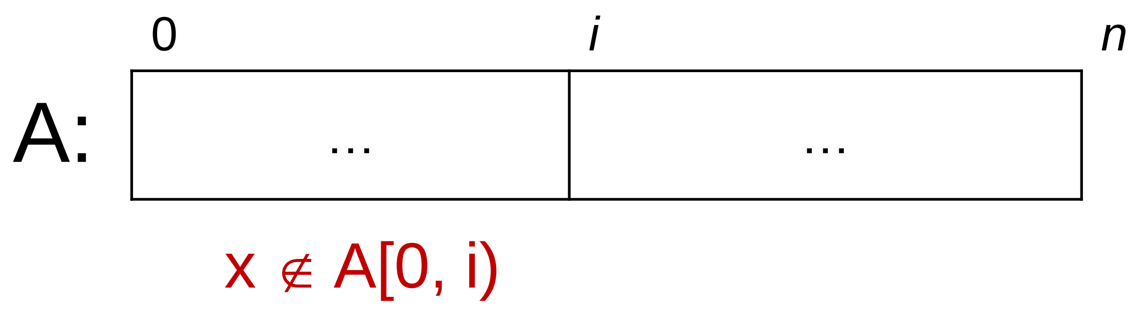
  math for !is_in(x, A, 0, n)

  - No point-to argument to do so!

```
1.  int search(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  /*@ensures (\result  == -1 && !is_in(x, A, 0, n))
4.                  || (0 <= \result && \result < n && A[\result] == x);
5.  @*/
6.  {
7.    for (int i = 0; i < n; i++)
8.    //@loop_invariant 0 <= i;
9.    {
10.     if (A[i] == x)  return i;
11.   }
12.   return -1;
13. }
```

Correctness (2)

- What do we know as we start iteration i of the loop?



  - $x \notin A[0, i)$
  - why?
    - Because we looked there and didn't find x

- This is something we believe to be true at every iteration of the loop
  - A loop invariant!
  - Well, a *candidate* loop invariant
    - We need to prove it is valid

Correctness (2)

return -1 on line 13

- **To show**: if $n = \text{\length}(A)$, then

  either $i = -1$ && $x \notin A[0, n)$

  or $0 \le i < n$ && $A[i] = x$

- We must prove
  1. $x \notin A[0, n)$
  2. $x \notin A[0, i)$ is a valid loop invariant

```
1.  int search(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  /*@ensures (\result  == -1 && !is_in(x, A, 0, n))
4.                  || (0 <= \result && \result < n && A[\result] == x);
5.  @*/
6.  {
7.    for (int i = 0; i < n; i++)
8.    //@loop_invariant 0 <= i;
9.    //@loop_invariant !is_in(x, A, 0, i);
10.   {
11.     if (A[i] == x)  return i;
12.   }
13.   return -1;
14. }
```

Correctness (2)

```
bool is_in(int x, int[] A, int lo, int hi)
//@requires 0 <= lo <= hi <= \length(A);
{
  if (lo == hi)  return false;
  return A[lo] == x || is_in(x, A, lo+1, hi);
}
```

$x \notin A[0, i)$ is a valid loop invariant

INIT:

- **To show**: $x \notin A[0, i)$ initially

A.	$i = 0$	by line 7
B.	$x \in A[0, 0)$ == false	by definition of is_in
C.	$x \notin A[0, i)$ == true	by math

✓

- A[0,0) is the empty array segment
  - Nothing is in it

```
1.  int search(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  /*@ensures …
4.              || …;
5.  @*/
6.  {
7.    for (int i = 0; i < n; i++)
8.    //@loop_invariant 0 <= i;
9.    //@loop_invariant !is_in(x, A, 0, i);
10.   {
11.     if (A[i] == x)  return i;
12.   }
13.   return -1;
14. }
```

```
bool is_in(int x, int[] A, int lo, int hi)
//@requires 0 <= lo <= hi <= \length(A);
{
  if (lo == hi)  return false;
  return A[lo] == x || is_in(x, A, lo+1, hi);
}
```

Correctness (2)

x ∉ A[0, i) is a valid loop invariant

PRES:

- **To show:** if x ∉ A[0, i), then x ∉ A[0, i')

A.	x ∉ A[0, i)	by assumption
B.	i' = i+1	by line 7
C.	x ∉ A[0, i+1) iff x ∉ A[0, i) and A[i] ≠ x	by def. of is_in

D. A[i] = x ??

a) If **true**: we return on line 11
   - We exit the function
   - We won't check the loop invariant again

b) If **false**: we continue with the loop
   - We will check the loop invariant again
   - x ∉ A[0, i+1)   by A, C, D(b)

✓

When returning from inside a loop, we don't need to show preservation

```
1.  int search(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  /*@ensures …
4.             || …;
5.  @*/
6.  {
7.    for (int i = 0; i < n; i++)
8.    ~~//@loop_invariant 0 <= i;~~
9.    //@loop_invariant !is_in(x, A, 0, i);
10.   {
11.     if (A[i] == x)  return i;
12.   }
13.   return -1;
14. }
```

Correctness (2)

return -1 on line 13

- We must prove
  - x ∉ A[0, i) is a valid loop invariant ✓
  - x ∉ A[0, n)

```
1.  int search(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  /*@ensures (\result  == -1 && !is_in(x, A, 0, n))
4.                   || (0 <= \result && \result < n && A[\result] == x);
5.  @*/
6.  {
7.    for (int i = 0; i < n; i++)
8.    //@loop_invariant 0 <= i;
9.    //@loop_invariant !is_in(x, A, 0, i);
10.   {
11.     if (A[i] == x)  return i;
12.   }
13.   return -1;
14. }
```

Correctness (2)

return -1 on line 13

- We must still prove $x \notin A[0, n)$
- When the loop terminates, we know that
  - $x \notin A[0, i)$ by line 9
  - $i \geq n$ by line 7

- To conclude $x \notin A[0, n)$ we need $i = n$

- Add $i \leq n$ as another loop invariant
  - Is it valid?

Left as exercise

```
1.  int search(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  /*@ensures …
4.               || …;
5.  @*/
6.  {
7.    for (int i = 0; i < n; i++)
8.    //@loop_invariant 0 <= i;
9.    //@loop_invariant !is_in(x, A, 0, i);
10.   {
11.     if (A[i] == x)  return i;
12.   }
13.   return -1;
14. }
```

Correctness (2)

return -1 on line 13

- We must still prove x ∉ A[0, n)

- When the loop terminates, we know that

A. x ∉ A[0, i)	by line 9
B. i ≥ n	by line 7
C. i ≤ n	by line 8
D. x ∉ A[0, n)	by A, B, C

✓

```
1.  int search(int x, int[] A, int n)
2.  //@requires n == \length(A);
3.  /*@ensures …
4.                 || …;
5.  @*/
6.  {
7.    for (int i = 0; i < n; i++)
8.    //@loop_invariant 0 <= i && i <= n;
9.    //@loop_invariant !is_in(x, A, 0, i);
10.   {
11.     if (A[i] == x)  return i;
12.   }
13.   return -1;
14. }
```

Final Code for seach

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i && i <= n;
  //@loop_invariant !is_in(x, A, 0, i);
  {
    if (A[i] == x)  return i;
  }
  return -1;
}
```

- We proved it safe and correct
- Does it do what we expect?
  - Yes!

Testing

Client View

- A caller of search can only rely on its contracts

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x);
@*/ ;
```

This is the **prototype** of this function

  - We may not be able to see the source code
    - it may have been written by someone else
    - it may be part of a library

- Can there be an implementation that satisfies these contracts but does not do what we expect?
    - An implementation that is correct, but wrong
  - Can there be **contract exploits**?

More Contract Exploits

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i && i <= n;
  //@loop_invariant !is_in(x, A, 0, i);
  {
    A[i] = x;                                    // puts x in A[0]
    if (A[i] == x)  return i;                    // and returns
  }
 return -1;
}
```

Even More Contract Exploits

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures (\result  == -1 && !is_in(x, A, 0, n))
                  || (0 <= \result && \result < n && A[\result] == x);
@*/
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i && i <= n;
  //@loop_invariant !is_in(x, A, 0, i);
  {
    A[i] = x + 1;                         // puts x+1 everywhere
    if (A[i] == x)  return i;             // will never return here
  }
 return -1;
}
```

Protecting against Contract Exploits

- The function changes the array
  - Caller has no way to know based on contracts

- What to do?
  - Even stronger contracts?
    - Check that the array doesn’t change
    - Cannot be done in C0
      - But other languages support this

  - **Unit testing**
    - Call search with a variety of inputs
      and check that it returns the expected value
    - Usually impractical to test with all possible inputs
      - Look for inputs where errors are likely

Testing C0 Functions

- Create a test file and write tests in its main function

- For each test
  - define input values
  - use **assert** to check that the function returns the expected result

- **assert**
  - aborts execution if its argument evaluates to **false**
  - continues with the next line if evaluates to **true**

  - **assert** is not a contract: we <u>cannot</u> use \length in it
  - //@assert is a contract: we <u>can</u> use \length in it

```
int main() {

  // Test #1
  int[] A = alloc_array(int, 2);
  A[0] = 3;
  A[1] = -7;

  assert(search(3, A, 2) == 0);
  assert(search(-7, A, 2) == 1);
  assert(search(42, A, 2) == -1);

  return 0;
}
```

Creates test array A = [3, -7]

3 is at index 0 of A: search(3, A, 2) should return 0

-7 is at index 1 of A

42 is not in A

Testing C0 Functions

- **Edge cases** are inputs at the edge of the input range
  - first element of an array
  - last element of an array
  - empty array
  - 1-element array

- Test as many edge cases as possible

```
int main() {
   …
  // Test #2
  int[] B = alloc_array(int, 4);
  for (int i=0; i<4; i++) B[i] = i+10;

  assert(search(10, B, 4) == 0);
  assert(search(13, B, 4) == 3);

 // Test #3
  int[] C = alloc_array(int, 0);
  assert(search(8, C, 0) == -1);

 // Test #4
  int[] D = alloc_array(int, 1);
  D[0] = 122;
  assert(search(122, D, 4) == 0);
  …
}
```

Creates test array B = [10, 11, 12, 13]

10 is the first element of B and 13 the last element

Nothing is in the empty array

Testing a 1-element array

Testing C0 Functions

- Test inputs that are easily mishandled
  - sorted arrays
    - with values that are
      - too small
      - too big
      - just right

```
int main() {
   …
  // Test #5
  int[] E = alloc_array(int, 6);
  for (int i=0; i<6; i++) E[i] = i+1;
  assert(search(-3, E, 6) == -1);
  assert(search(4, E, 6) == 3);
  assert(search(9, E, 6) == -1);

 // Test #6
  int[] F = alloc_array(int, 6);
  for (int i=0; i<6; i++) F[i] = 6-i;
  assert(search(-3, F, 6) == -1);
  assert(search(4, F, 6) == 2);
  assert(search(9, F, 6) == -1);
  …
}
```

E is the sorted array
E = [1, 2, 3, 4, 5, 6]

F is E in
reverse order

Testing C0 Functions

- For good measure, include some big inputs and test them systematically
  - these are called **stress tests**

For big tests, putting the size in a variable makes it easy to modify

G contains the first n even numbers

G[i] contains 2*i

G contains no odd number

```
int main() {
    …
 // Test #7
  int n = 1000000;
  int[] G = alloc_array(int, n);
  for (int i=0; i<n; i++) G[i] = 2*i;

  for (int i=0; i<n; i++)
    assert(search(2*i, G, n) == i);

  for (int i=0; i<2*n; i++)
    assert(search(2*i + 1, G, n) == -1);
  …
}
```

  - best would be to use random inputs
    - ➢ we will see later how to do that

Testing C0 Functions

- **Do not** test implementation details
  - anything that the function description leaves open-ended

H is initialized with the default int
H = [0, 0, 0, 0, 0]

BAD TEST

```
int main() {
   …
  // Test #8
  int E = alloc_array(int, 5);
  assert(search(0, E, 5) == 0);

  return 0;
}
```

- Example: *array with duplicate elements*
  - nothing tells us the index of which occurrence search will return
    - our implementation returns the first
    - but other implementations may return
      - the last
      - the middle occurrence
      - a random occurrence
      - …